

BY JULIA EVANS

10

10

SELECT author, COUNT(\*)  
FROM posts  
GROUP BY author

SELECT  
FROM  
WHERE  
GROUP BY  
HAVING  
LIMIT  
ORDER BY  
SELECT  
+



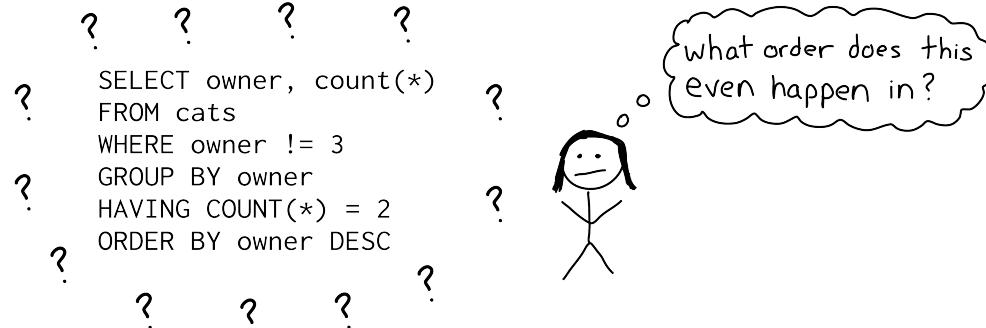
SELECT STAR  
BECOME A

\*wizardzines.com \*

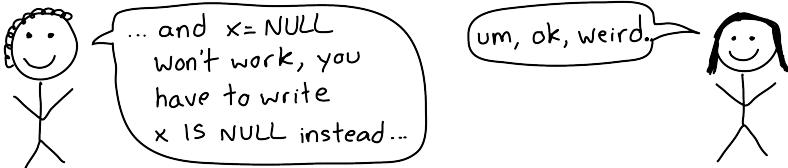
more zines at  
love this?

# about this zine

SQL isn't like most programming languages.



and it has quite a few weird gotchas:



but knowing it lets you use really powerful database software to answer questions FAST:



This zine will get you started with SELECT queries so you can get the answers to any question you want about your data. ❤️ You can run any query from the zine here:

<https://sql-playground.wizardzines.com>

## ♥ thanks for reading ♥

When you're learning it's important to experiment! So you can try out any of the queries in this zine and run your own in an SQL playground:

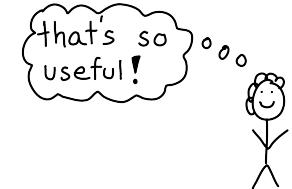
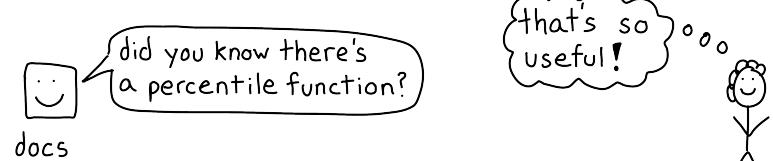
★ <https://sql-playground.wizardzines.com> ★

Here are a few more great SQL resources:

★ Use the index, Luke! (<https://use-the-index-luke.com>) is a very in-depth explanation of how to use indexes to make your queries fast.

★ There are several visualizers that will help you understand the output of an EXPLAIN. For example: <https://explain.depesz.com/> for PostgreSQL!

★ The official documentation is always GREAT for learning about SQL functions.



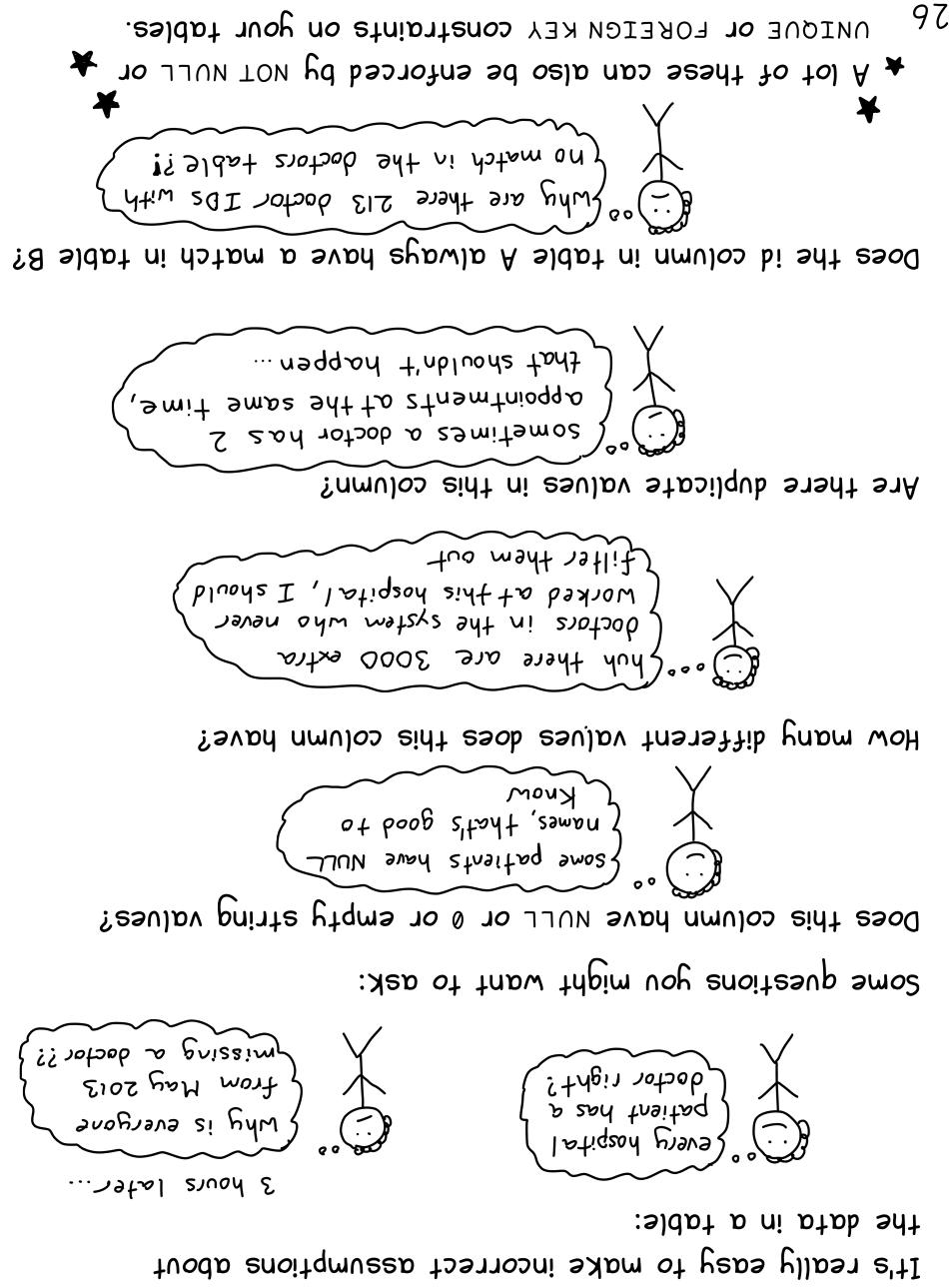
### ★ credits ★

Cover illustration: Deise Lino

Editing: Dolly Lanuza, Kamal Marhubi, Samuel Wright  
Reviewers: Anton Dubrau, Arielle Evans

4	gettting started with SELECT
5	SELECT queries start with FROM
6	SELECT
7	WHERE
8	GROUP BY
9	HAVING
10-11	JOINS
12	example: GROUP BY
13	ORDER BY + LIMIT
14	intro
15	OVER() assigns every row a window
16	much ado about NULL
17	NULL: unknown or missing
18	NULL surprises
19	handle NULLS with COALESCE
20	CASE
21	ways to count rows
22	subqueries
23	tip: single quote strings
24	how indexes make your queries fast
25	EXPLAIN your slow queries
26	questions to ask about your data
27	thanks for reading ♡

# Table of Contents

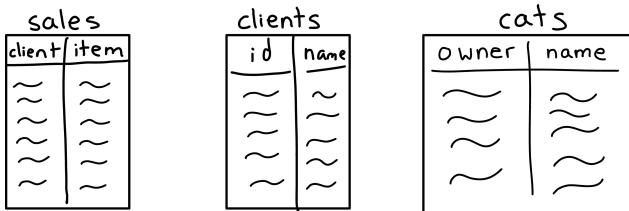


about your data  
questions to ask

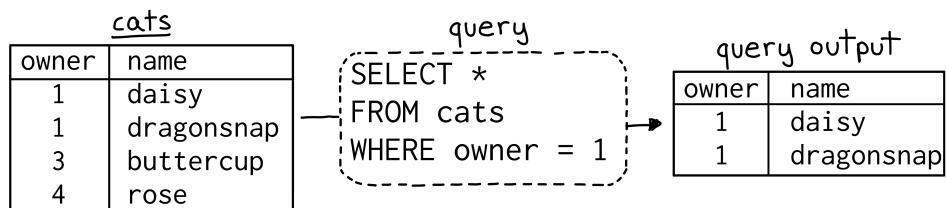
about your data  
questions to ask

# getting started with SELECT

A SQL database contains a bunch of tables



Every SELECT query takes data from those tables and outputs a table of results.



A few basic facts to start out:

→ SELECT queries have to be written in the order:

SELECT ... FROM ... WHERE ... GROUP BY .. HAVING ...  
ORDER BY ... LIMIT

→ SQL isn't case sensitive: select \* from table is fine too.  
This zine will use ALL CAPS for SQL keywords like FROM.



there are other kinds of  
queries like INSERT/UPDATE/DELETE  
but this zine is just about SELECT

# EXPLAIN your slow queries

Sometimes queries run slowly, and EXPLAIN can tell you why!

2 ways you can use EXPLAIN in PostgreSQL: (other databases have different syntax for this)

① Before running the query (EXPLAIN SELECT ... FROM ...)

This calculates a query plan but doesn't run the query.

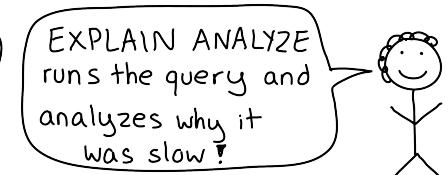


I always run EXPLAIN on a query before running it on my production database. I won't risk overloading the database with a slow query!

② After running the query (EXPLAIN ANALYZE SELECT ... FROM ...)



why is my query so slow?



EXPLAIN ANALYZE runs the query and analyzes why it was slow!

Here are the EXPLAIN ANALYZE results from PostgreSQL for the same query run on two tables of 1,000,000 rows: one table that has an index and one that doesn't

EXPLAIN ANALYZE SELECT \* FROM users WHERE id = 1

unindexed table

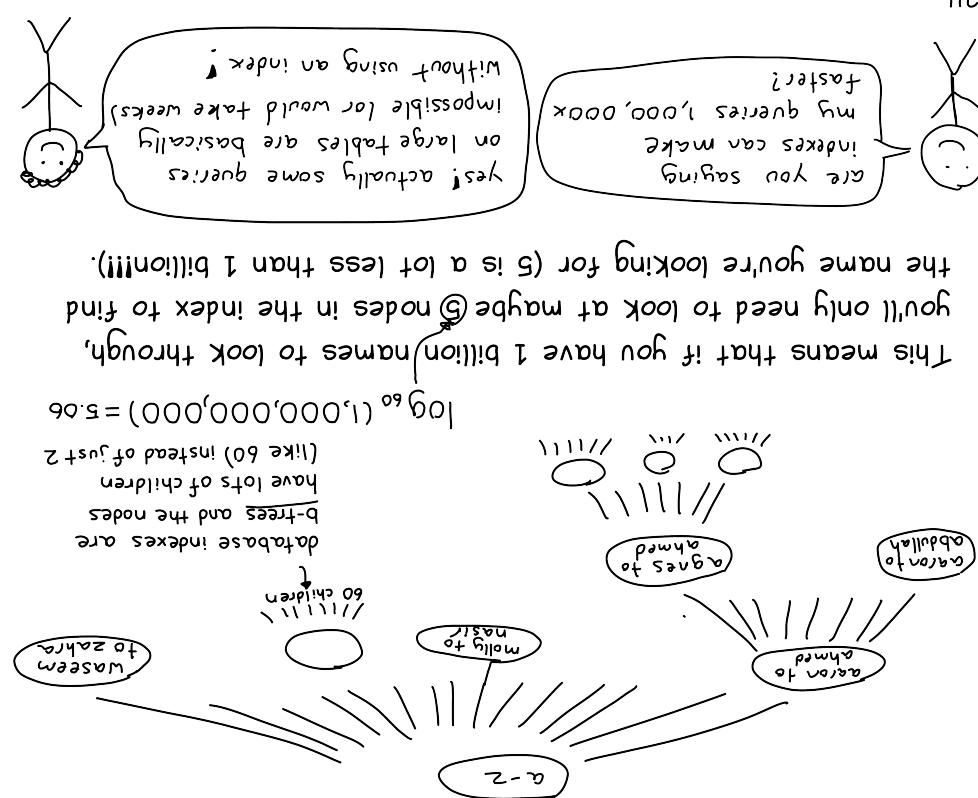
Seq Scan on users  
Filter: (id = 1)  
Rows Removed by Filter: 999999  
Planning time: 0.185 ms  
Execution time: 179.412 ms

indexed table

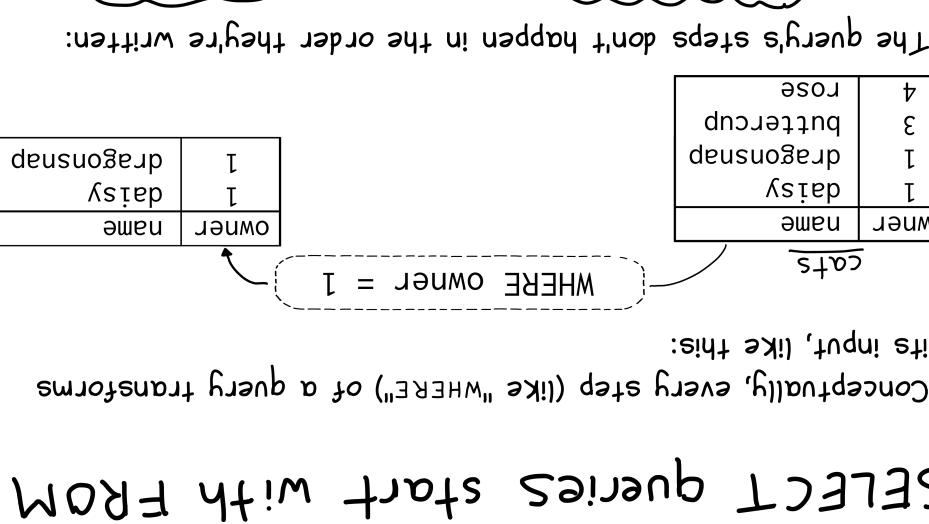
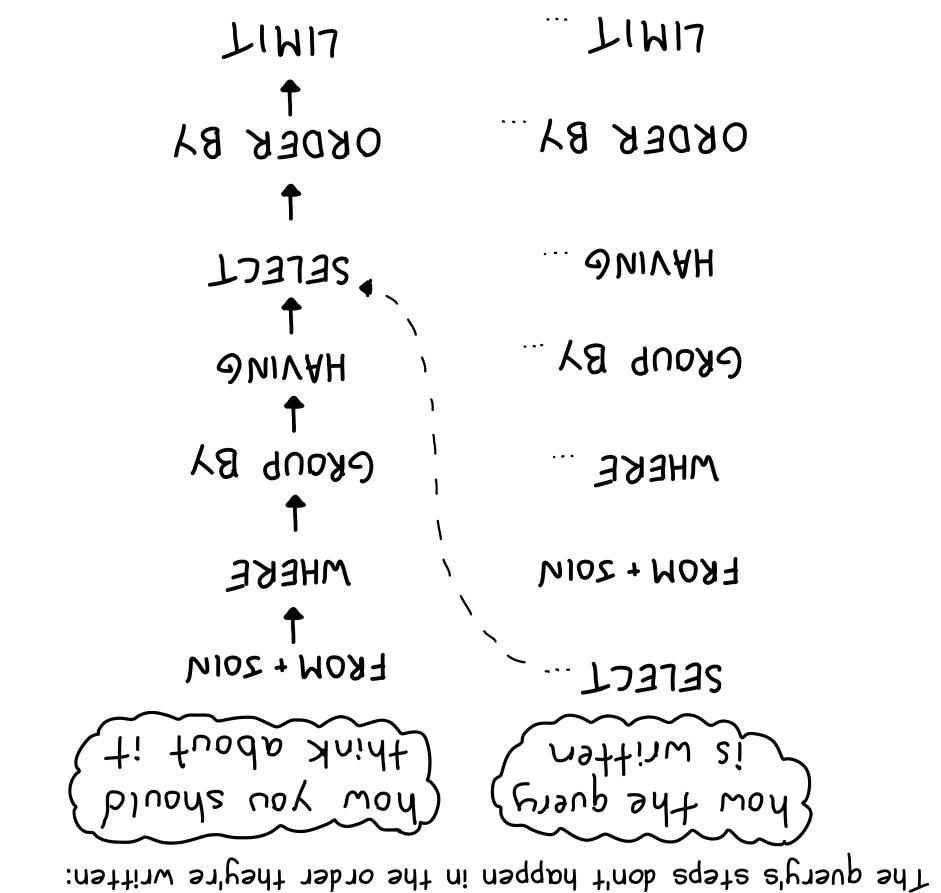
Index Only Scan using users\_id\_idx on users  
Index Cond: (id = 1)  
Heap Fetches: 1  
Planning time: 3.411 ms  
Execution time: 0.088 ms

"Seq Scan" means it's looking at each row (slow!)

the query runs ~~50~~  
times faster with an index



(In reality query execution is much more complicated than this.)  
There are a lot of optimizations.)



# SELECT

SELECT is where you pick the final columns that appear in the table the query outputs. Here's the syntax:

```
SELECT expression_1 [AS alias],  
      expression_2 [AS alias2],  
  FROM ...
```

Some useful things you can do in a SELECT:

## ★ Combine many columns with SQL expressions

A few examples:

```
CONCAT(first_name, ' ', last_name)  
DATE_TRUNC('month', created) ← this is PostgreSQL syntax for  
rounding a date, other SQL  
dialects have different syntax
```

## ★ Alias an expression with AS

↑ is a concatenation operation  
first\_name || ' ' || last\_name is a mouthful! If you alias  
an expression with AS, you can use the alias elsewhere in  
the query to refer to that expression.

```
SELECT first_name || ' ' || last_name AS full_name  
FROM people  
ORDER BY full_name DESC  
↑ refers to first_name || ' ' || last_name
```

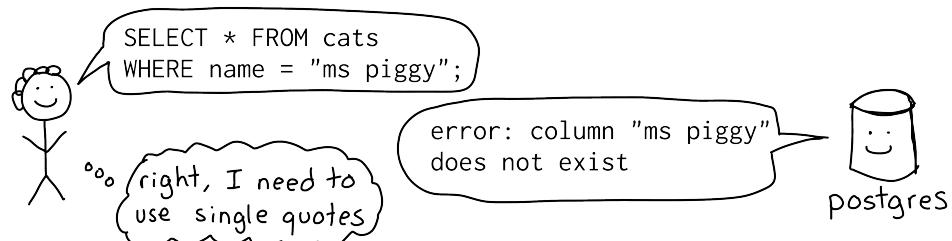
## ★ Select all columns with SELECT \*

When I'm starting to figure out a query, I'll often write  
something like

```
SELECT * FROM some_table LIMIT 10  
just to quickly see what the columns in the table look like.
```

## tip: single quote strings

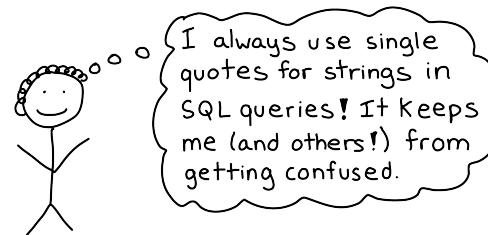
In some SQL implementations (like PostgreSQL), if you double quote a string it'll interpret it as a column name:



Here's a table explaining what different quotes mean in different SQL databases. "Identifier" means a column name or table name.

Sometimes table names have special characters like spaces in them so it's useful to be able to quote them.

	single quotes 'ms piggy'	double quotes "ms piggy"	backticks '`ms piggy`'
MySQL	string	string or identifier	identifier
PostgreSQL	string	identifier	invalid
SQLite	string	string or identifier	identifier
SQL server	string	string or identifier	invalid





# GROUP BY

GROUP BY combines multiple rows into one row. Here's how it works for this table & query:

SELECT item, COUNT(\*), MAX(price)  
FROM sales  
GROUP BY item

*aggregates*

sales	item	price
catnip	5	
laser	8	
tuna	4	
tuna	3	

query output	item	COUNT(*)	MAX(price)
catnip	1	5	
laser	1	8	
tuna	2	4	

① Split the table into groups for each value that you grouped by:

item='catnip'	item='laser'	item='tuna'
item   price	item   price	item   price
catnip   5	laser   8	tuna   4
		tuna   3

② Calculate the aggregates from the query for each group:

item	price	item	price	item	price
catnip	5	laser	8	tuna	4
COUNT(*) = 1	COUNT(*) = 1	COUNT(*) = 2			
MAX(price) = 5	MAX(price) = 8	MAX(price) = 4			

③ Create a result set with 1 row for each group

item	COUNT(*)	MAX(price)
catnip	1	5
laser	1	8
tuna	2	4

# ways to count rows

Here are three ways to count rows:

① COUNT(\*) : count all rows

This counts every row, regardless of the values in the row. Often used with a GROUP BY to get common values, like in this "most popular names" query:

```
SELECT first_name, COUNT(*)
FROM people
GROUP BY first_name
ORDER BY COUNT(*) DESC
LIMIT 50
```

② COUNT(DISTINCT column): get the number of distinct values

Really useful when a column has duplicate values. For example, this query finds out how many species every plant genus has:

"GROUP BY 1" means group by the first expression in the SELECT

```
SELECT genus, COUNT(DISTINCT species)
FROM plants
GROUP BY 1
ORDER BY 2 DESC
```

③ SUM(CASE WHEN expression THEN 1 ELSE 0 END)

This trick using SUM and CASE lets you count how many cats vs dogs vs other animals each owner has:

I like to put commas at the start for big queries

SELECT owner
, SUM(CASE WHEN type = 'dog' then 1 else 0 end) AS num_dogs
, SUM(CASE WHEN type = 'cat' then 1 else 0 end) AS num_cats
, SUM(CASE WHEN type NOT IN ('dog', 'cat') then 1 else 0 end) AS num_other
FROM pets GROUP BY owner

*pets*

owner	type
1	dog
1	cat
2	dog
2	parakeet

owner	num_dogs	num_cats	num_other
1	1	1	0
2	1	0	1

# HAVING

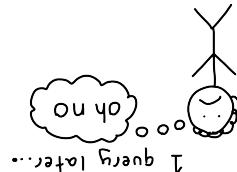
This query uses HAVING to find all emails that are shared by more than one user:

```

SELECT email, COUNT(*)
FROM users
GROUP BY email
HAVING COUNT(*) > 1

```

A diagram showing two user icons connected to a single cloud-like shape containing the text "email". A thought bubble above them says "every user has a different email right?"



id	email	COUNT(*)
1	email1	1
2	asdfefake.com	2
3	bobgenuine.com	3

query output

ROWS AFTER grouping and WHERE filters rows BEFORE grouping. HAVING is like WHERE, but with 1 difference: HAVING filters because of this, you can use aggregates (like COUNT(\*)) in a HAVING clause but not WITH WHERE.

Here's another HAVING example that finds months with more than \$6.00 in income:

SELECT month  
FROM sales  
GROUP BY month  
HAVING SUM(price) > 6

month	item	price	sales
Jan	laser	8	
Feb	catnip	5	
March	food	4	
April	lasers	3	

query output

CASE is how to write an if statement in SQL. Here's the syntax:

```

CASE
WHEN <Condition> THEN <result>
WHEN <other-condition> THEN <result>
...
ELSE <result>
END

```

Here's how to categorize people into age ranges!

# ★ EXAMPLE ★

first-name	age	age-range
ahmed	5	child
marie	17	teenager
akira	60	adult
pablo	15	teenager

people
ahmed
marie
akira
pablo
15
60
17
ahmed
marie
akira
pablo

SELECT first-name, age, CASE  
WHEN age < 13 THEN 'child',  
WHEN age < 20 THEN 'teenager',  
ELSE 'adult', END AS age-range  
FROM people

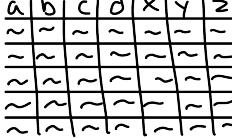


Often I want to categorize by something that isn't a column:

# \* my rules for simple JOINS \*

Joins in SQL let you take 2 tables and combine them into one.

a	b	c	d	x	y	z
~	~	~	~	~	~	~
~	~	~	~	~	~	~
~	~	~	~	~	~	~
~	~	~	~	~	~	~
~	~	~	~	~	~	~
~	~	~	~	~	~	~
~	~	~	~	~	~	~

INNER JOIN         $\Rightarrow$     

Joins can get really complicated, so we'll start with the simplest way to join. Here are the rules I use for 90% of my joins:

Rule 1: only use LEFT JOIN and INNER JOIN

There are other kinds of joins (RIGHT JOIN, CROSS JOIN, FULL OUTER JOIN), but 95% of the time I only use LEFT JOIN and INNER JOIN.

Rule 2: refer to columns as table\_name.column\_name

You can leave out the table name if there's just one column with that name, but it can get confusing

Rule 3: Only include 1 condition in your join

Here's the syntax for a LEFT JOIN:

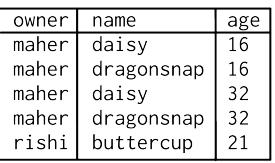
table1 LEFT JOIN table2 ON <any boolean condition>  
I usually stick to a very simple condition, like this:

```
table1 LEFT JOIN table2  
ON table1.some_column = table2.other_column
```

Rule 4: One of the joined columns should have unique values

If neither of the columns is unique, you'll get strange results like this:

owners_bad		cats_bad		owners_bad INNER JOIN cats_bad ON owners_bad.name = cats_bad.owner		
name	age	owner	name	owner	name	age
maher	16	maher	daisy	maher	daisy	16
maher	32	maher	dragonsnap	maher	dragonsnap	16
rishi	21	rishi	buttercup	maher	daisy	32
				maher	dragonsnap	32
				rishi	buttercup	21

INNER JOIN     $\Rightarrow$     

(these are "bad" versions of the owners and cats tables that don't JOIN well!)

# handle NULLS with COALESCE

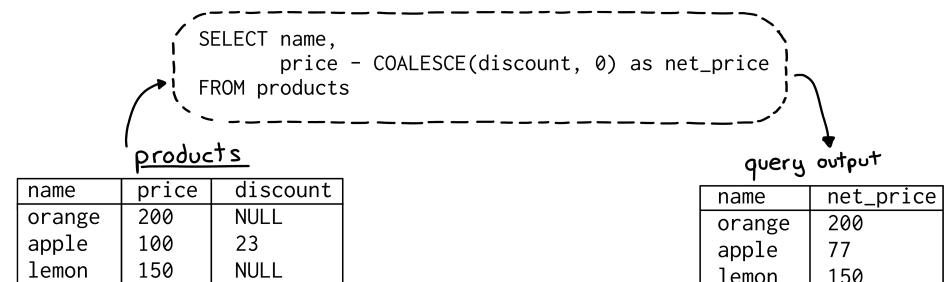
COALESCE is a function that returns the first argument you give it that isn't NULL

COALESCE(NULL, 1, 2) => 1  
COALESCE(NULL, NULL, NULL) => NULL  
COALESCE(4, NULL, 2) => 4

2 ways you might want to use COALESCE in practice:

① Set a default value

In this table, a NULL discount means there's no discount, so we use COALESCE to set the default to 0:



products

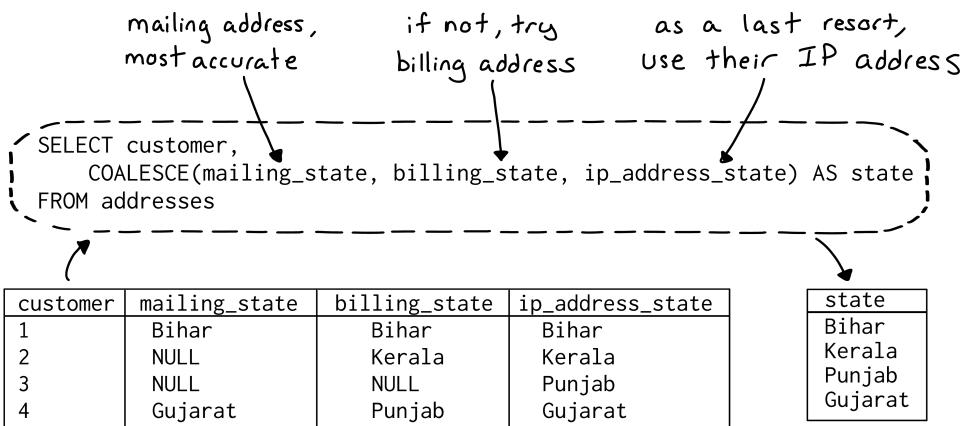
name	price	discount
orange	200	NULL
apple	100	23
lemon	150	NULL

query output

name	net_price
orange	200
apple	77
lemon	150

② Use data from 2 (or more!) different columns

This query gets the best guess at a customer's state:



mailing address,  
most accurate

if not, try  
billing address

as a last resort,  
use their IP address

addresses

customer	mailing_state	billing_state	ip_address_state
1	Bihar	Bihar	Bihar
2	NULL	Kerala	Kerala
3	NULL	NULL	Punjab
4	Gujarat	Punjab	Gujarat

state

	Bihar	Kerala	Punjab	Gujarat
1				
2				
3				
4				

# NULL SURPRISES

More operations with NULL which might be surprising: NULL isn't even equal to itself!

```

    2 + NULL => NULL
    2 = NULL => NULL
    NULL = NULL => NULL
    2 != NULL => NULL
    CONCAT('hi', NULL) => NULL
  
```

## more surprising truths

To match NULLs as well, I'll often write something like WHERE name != 'betty', OR name IS NULL instead.

SELECT * FROM fish	WHERE name != 'betty'	name owner ahmed
		name bob nemo

fish

SURPRISE! name != 'betty', doesn't match NULLs

works

```

  name IS NOT NULL
  name=NULL
  name != NULL
  name IS NULL
  
```

doesn't work

SELECT * FROM fish	WHERE name = NULL	name owner ahmed
		name bob nemo

fish

SURPRISE! x = NULL doesn't work

that are surprising at first:

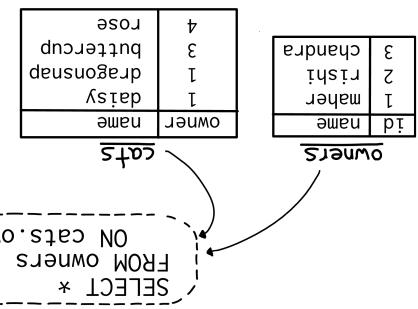
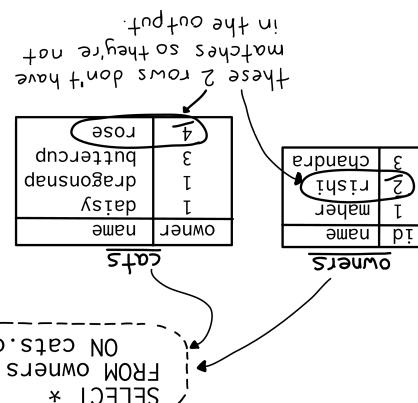
$x \neq \text{NULL}$  are never true for any  $x$ ). This results in 2 behaviors that are surprising at first:

$x \neq \text{NULL}$  (or not equal) to anything in SQL ( $x = \text{NULL}$  and

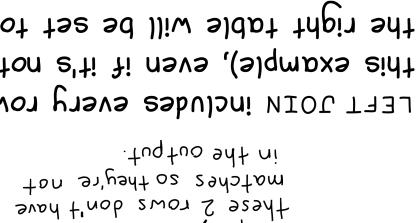
INNER JOIN only includes rows that match the ON condition.

Here are examples of how INNER JOIN and LEFT JOIN work:

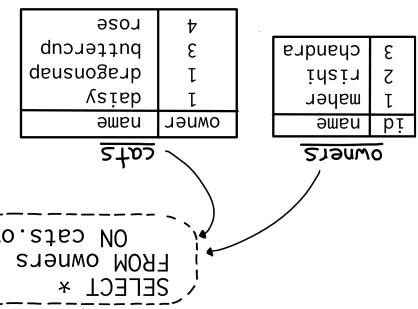
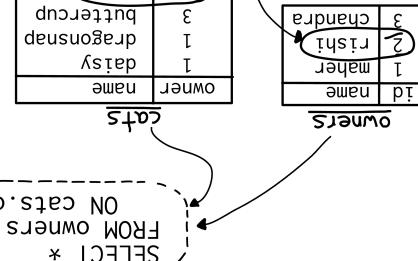
This query combines the cats and owners tables:



LEFT JOIN includes every row from the left table (owners in this example), even if it's not in the right table. Rows not in the right table will be set to NULL.



LEFT JOIN includes every row from the left table (owners in this example), even if it's not in the right table. Rows not in the right table will be set to NULL.

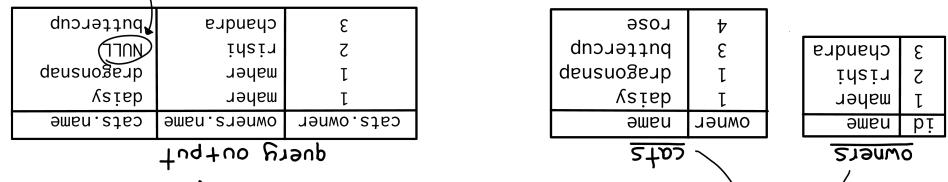


LEFT JOIN includes every row from the left table (owners in this example), even if it's not in the right table. Rows not in the right table will be set to NULL.

- 1) it's an INNER JOIN / LEFT JOIN
- 2) it includes the table name in cats.owner and owners.id
- 3) the condition ON cats.owner = owners.id is simple
- 4) it joins on a unique column (the id column in the owners table)

from the previous page:

This is a classic example of a join that follows my guidelines:



SELECT \* FROM owners INNER JOIN cats  
ON cats.owner = owners.id

# example : LEFT JOIN + GROUP BY

This query counts how many items every client bought

(including clients who didn't buy anything):

```
SELECT name, COUNT(item) AS items_bought
FROM owners LEFT JOIN sales
    ON owners.id = sales.client
GROUP BY name
ORDER BY items_bought DESC
```

FROM owners LEFT JOIN sales...

owners	
id	name
1	maher
2	rishi
3	chandra

sales	
item	client
catnip	1
laser	1
tuna	1
tuna	2

ON owners.id = sales.client

GROUP BY name

id	name	item
1	maher	catnip
1	maher	laser
1	maher	tuna
2	rishi	tuna
3	chandra	NULL

id	name	item
1	maher	catnip
1	maher	laser
1	maher	tuna
2	rishi	tuna
3	chandra	NULL

SELECT name, COUNT(item)  
AS items\_bought

ORDER BY  
items\_bought DESC

name	items_bought
rishi	1
chandra	3
maher	0

COUNT(item)  
doesn't count  
NULLs

name	items_bought
maher	3
rishi	1
chandra	0

# NULL: unknown or missing

NULL is a special state in SQL. It's very commonly used as a placeholder for missing data ("we don't know her address!")

What NULL means exactly depends on your data. For example, it's really important to know if allergies IS NULL means

→ "no allergies" or

→ "we don't know if she has allergies or not" always

NULL "should" mean  
"unknown" but it doesn't



it would be way easier  
if NULL always meant  
the same thing but it  
really depends on  
your data!

## ★ Where NULLs come from ★

→ There were already NULL values in the table

→ The window function LAG() can return NULL

→ You did a LEFT JOIN and some of the rows on the left didn't have a match for the ON condition

ooh, not every cat has  
an owner so sometimes  
the owner name is NULL

## ★ Ways to handle NULLs ★

→ Leave them in!

I'd rather see a NULL  
and know there's missing data  
than get misleading results

→ Filter them out!

... WHERE first\_name IS NOT NULL ...

→ Use COALESCE or CASE to add a default value

owner	name
4	daisy
1	rose
4	lily

owner	name
1	daisy
1	dragonsnap
3	buttercup

SELECT \* FROM cats  
ORDER BY LENGTH(name) ASC  
LIMIT 2

For example, this is the same as the previous query, but it limits to only the 2 cats with the shortest names:

## LIMIT [integer]

The syntax is:

LIMIT lets you limit the number of rows output.

owner	name
4	rose
1	daisy
3	dragonsnap

owner	name
1	daisy
1	dragonsnap
3	buttercup

SELECT \* FROM cats  
ORDER BY LENGTH(name) ASC  
LIMIT 2

For example, this query sorts cats by the length of their name (shortest first):

## ORDER BY [expression] ASC

ORDER BY lets you sort by anything you want! The syntax is:

ORDER BY and LIMIT happen at the end and affect the final output of the query.

## ORDER BY and LIMIT

event	hour	time_since_last
1	NULL	LAG(1)
2	NULL	in the first row
3	is Null for	the window
4	feeding	daisy
5	diaper	dragonsnap
6	diaper	buttercup
7	diaper	rose

## ⑤ ORDER BY hour ASC

### ④ SELECT + type, hour, HOUR - LAG(hour)

event	hour	time_since_last
1	NULL	LAG(1)
2	NULL	in the first row
3	is Null for	the window
4	feeding	daisy
5	diaper	dragonsnap
6	diaper	buttercup
7	diaper	rose

## ③ OVER (PARTITION BY event ORDER BY hour ASC)

event	hour	time_since_last
1	1	diaper
2	1	diaper
3	1	diaper
4	1	diaper
5	5	feeding
6	5	feeding
7	5	feeding

## ② WHERE event IN ('diaper', 'feeding')

### ① FROM baby-log

ORDER BY hour ASC

WHERE event in ('feeding', 'diaper')

FROM baby-log

AS time\_since\_last

SELECT event, hour, LAG(hour) OVER(PARTITION BY event ORDER BY hour ASC)

change.

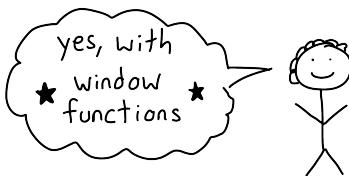
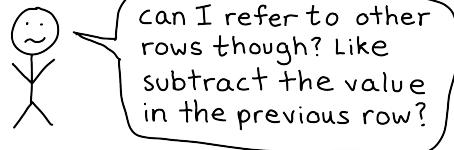
This query finds the time since a baby's last feeding/diaper change.  
example: get the time between baby feedings

# refer to other rows with ★ window functions ★

Let's talk about an ~~advanced~~ SQL feature: window functions!  
Normally SQL expressions only let you refer to information in  
a single row.

2 columns from the same row

```
SELECT CONCAT(firstname, ' ', lastname) as full_name
```



Window functions are SQL expressions that let you reference values in other rows. The syntax (explained on the next page!) is:

[expression] OVER ([window definition])

Example: use LAG() to find how long since the last sale

item	day
catnip	2
laser	40
tuna	70
tuna	72

`SELECT item, day - LAG(day) OVER (ORDER BY day)  
FROM sales`

query output

item	day - LAG(day) OVER (ORDER BY day)
catnip	NULL ← 2-NULL
laser	38 ← 40-2
tuna	30 ← 70-40
tuna	2 ← 72-70

They're part of SELECT, so they happen after HAVING:

FROM + JOIN → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT



# OVER() assigns every row a window

A "window" is a set of rows.

name	class	grade
juan	1	93
lucia	1	98

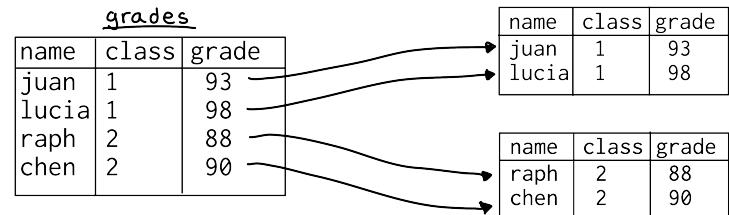
a window!

A window can be as big as the whole table (an empty OVER() is the whole table!) or as small as just one row.

OVER() is confusing at first, so here's an example! Let's run this query that ranks students in each class by grade:

```
SELECT name, class, grade,  
ROW_NUMBER() OVER (PARTITION BY class  
ORDER BY grade DESC)  
AS rank_in_class  
FROM grades
```

Step 1: Assign every row a window. OVER(PARTITION BY class) means that there are 2 windows: one each for class 1 and 2



Step 2: Run the function. We need to run ROW\_NUMBER() to find each row's rank in its window:

name	class	grade	rank_in_class
juan	1	93	2
lucia	1	98	1
raph	2	88	2
chen	2	90	1